EFFICIENTLY RELEASING LOCKS WHEN AN EXCEPTION OCCURS

FIELD OF THE INVENTION

The present invention relates to compilers for programming languages and, in particular, to compiling programming code such that locks are efficiently released when an exception occurs during execution of the code.

BACKGROUND OF THE INVENTION

The Java programming language is recognized to provide many benefits to the programmer. Not the least of these benefits relate to the handling of error conditions, support for multiple threads (to be defined hereinafter) and platform independence.

A defined unit of programming code, developed for a particular purpose, has been called a function, a subroutine and a procedure in different programming languages. In the Java programming language, such a unit is called a "method".

Java includes provisions for handling unusual error conditions. The provisions are included in the form of "exceptions", which allow unusual error conditions to be handled without including "if" statements to deal with every possible error condition.

Java also includes provisions for multiple executions streams running in parallel. Such executions streams are called "threads".

One of the desirable qualities of Java, is the ability of Java code to be executed on a wide range of computing platforms, where each of the computing platforms in the range can run a Java Virtual Machine. However, there remains a requirement that the code run on a particular platform be native to that platform. A just-in-time (JIT) Java

compiler is typically in place, as part of an environment in which Java code is to be executed, to convert Java byte code into native code for the platform on which the code is to be executed.

Recent JIT compilers have improved features, such as "method inlining" and "synchronization".

Where a method is called that does something trivial, like add one to an argument and then return the argument, Java programmers have, in the past, been tempted to simply insert the instruction rather than call the method. This act is called manual method inlining. Such manual method inlining can improve the speed at which the code runs as the overhead of an instruction that jumps to the called method, as well as the return from the method, is saved. Some JIT compilers have the capability to recognize where method inlining will improve the speed at which code runs and, thus, can automatically inline methods while converting the byte code into native code.

From Venners, Bill, "How the Java virtual machine performs thread synchronization", http://www.javaworld.com/javaworld/jw-07-1997/jw-07-hood.html, in the Java virtual machine (JVM), each thread is awarded a Java stack, which contains data no other thread can access. If multiple threads need to use the same objects or class variables concurrently, the access of the threads to the data must be properly managed. Otherwise, the program will have unpredictable behavior.

To coordinate shared data access among multiple threads, the Java virtual machine associates a lock with each object. A thread needing to lock a particular object, communicates this requirement to the JVM. The JVM may then provide the lock to the thread. When the thread no longer requires the lock, the thread communicates this lack of requirement to the JVM. If a second thread has requested the same lock, the JVM provides the lock to the second thread.

A single thread is allowed to lock the same object multiple times. For each object, the JVM maintains a count of the number of times the object has been locked. An unlocked object has a count of zero. When a thread acquires the lock for the first time, the count is incremented to one. Each time the thread acquires a lock on the same object, the count is incremented. Each time the thread releases the lock, the count is decremented. When the count reaches zero, the lock is released and made available to other threads.

In Java language terminology, the coordination of multiple threads that must access shared data is called synchronization. The Java language provides two built-in ways to synchronize access to data: with synchronized statements or synchronized methods.

Two bytecodes, namely "monitorenter" and "monitorexit", are used for synchronization blocks within methods. When the monitorenter bytecode is encountered by the Java virtual machine, the Java virtual machine acquires the lock for the object referred to by objectref on the stack. If the thread already owns the lock for that object, a count is incremented. Each time the monitorexit bytecode is executed for the thread on the object, the count is decremented. When the count reaches zero, the lock is released.

When an exception occurs in an executing Java method, any locks on the objects with which the method has been interacting must be released. As will be apparent to a person skilled in the art, the locks that must be released do not include the locks that are held at entry to the handler's try region. In a known JIT compiler, monitorenter and monitorexit locking operations use a "try region" and a handler to guarantee that an object is unlocked even if an exception occurs. Such a handler effectively catches and re-throws exceptions to guarantee that each locked object is unlocked. Such an approach may be considered costly in terms of execution time required to throw and handle multiple exceptions.

SUMMARY OF THE INVENTION

In compiling the program code for a method, synchronization depths are recorded in a table for various ranges of program counter addresses in the method. A portion of a stack frame associated with the method may be dedicated for the recordation of references to objects locked during the execution of the compiled method code. Such references are recorded in the stack frame at a location based on synchronization depth. When an exception occurs, the synchronization depth may be determined from the table and used to obtain a reference to an object from which a lock should be removed. Advantageously, the costly practice of catching and re-throwing exceptions is avoided and, thus, execution time may be shortened.

In accordance with an aspect of the present invention there is provided a method of handling an exception. The method includes recognizing that an exception has occurred during the execution of a given method, consulting a stack frame associated with the given method to determine an identity of an object required to be unlocked and removing the lock on the object. In other aspects of the present invention, an exception handler is provided for performing this method and a computer readable medium is provided to allow a general purpose computer to perform this method.

In accordance with another aspect of the present invention there is provided, at a just in time compiler of programming language code, the programming language code including a plurality of instructions, a method of generating executable code. The method includes determining a synchronization depth for each instruction, associating the synchronization depth with a program counter address associated with each instruction, determining a continuous range of program counter addresses associated with an equivalent synchronization depth and storing an indication of the continuous range of program counter addresses in a table associated with the equivalent synchronization depth. In other aspects of the present invention, a just in time compiler

is provided for performing this method and a computer readable medium is provided to allow a general purpose computer to perform this method.

Other aspects and features of the present invention will become apparent to those of ordinary skill in the art upon review of the following description of specific embodiments of the invention in conjunction with the accompanying figures.

BRIEF DESCRIPTION OF THE DRAWINGS

In the figures which illustrate example embodiments of this invention:

FIG. 1 illustrates an exemplary code fragment for use in describing the operation of the present invention;

FIG. 2 illustrates a method of the exemplary code of FIG. 1, wherein several methods have been inlined;

FIG. 3 illustrates a compiled code listing corresponding to the inlining version of the code of FIG. 1 as output from a typical JIT compiler;

FIG. 4 illustrates a synchronization depth table for the method illustrated in FIG. 2, constructed according to an aspect of the present invention;

FIG. 5 illustrates a stack frame related to the method illustrated in FIG. 2;

FIG. 6 illustrates an inlining tree related to the method illustrated in FIG. 2;

FIG. 7 illustrates a simplified version of the code of the method of FIG. 2, only showing synchronization operations;

FIG. 8 illustrates steps in program code compiling method according to an aspect of the present invention; and

FIG. 9 illustrates steps in an error handling method according to an aspect of the present invention.

DETAILED DESCRIPTION

An exemplary code fragment 100 in FIG. 1 has five methods, including inner_foo() on lines 102-104, foo() on lines 105-107, bar() on lines 108-110, go_to_it() on lines 111- 123 and main() on lines 124-127. Since the go_to_it() method calls the foo(), inner_foo() and bar() methods in a try block defined over lines 114-118, a JIT compiler may recognize that the called methods may be inlined into the go_to_it() method. Such inlining creates two synchronized regions in the try block, as shown in an inlining version 200 of the go_to_it() method illustrated in FIG. 2.

In the inlining version of the go_to_it() method 200, lines 207-209 may be recognized as the inlined inner_foo() method, lines 207-209 may be recognized as the inlined foo() method and lines 213-215 may be recognized as the inlined bar() method. If the code on one of the lines 207-209 causes an exception, then a receiver object must be unlocked twice before the catch block (lines 216-218) may be executed. Similarly, if the code on lines 213-215 causes an exception, then the receiver object must be unlocked once before the catch block (lines 216-218) may be executed.

A javac compiler (a known Java compiler) compiling the inlining version of go_to_it() 200, which is directly written in Java using the synchronized statement, would insert an artificial try region around each synchronized code region (lines 207-209 and lines 213- 215). Such an artificial try region has a catch block that unlocks the object before re- throwing the exception. The synchronization would typically be implemented using the Java bytecodes monitorenter and monitorexit. FIG. 3 illustrates a compiled

code listing 300 corresponding to the inlining version of go_to_it() 200 as output from a typical JIT compiler. The artificial try regions and monitorenter and monitorexit bytecodes should be evident in the compiled code listing 300.

In contrast, where the inlining version of the go_to_it() method 200 is compiled by a JIT compiler employing aspects of the present invention to inline the foo() and bar() methods, no artificial try regions are inserted and the synchronized regions may be implemented using artificial, herein-proposed "syncenter" and "syncexit" bytecodes.

The introduction of these artificial bytecodes has two main benefits. First, unlike the more general monitorenter and monitorexit bytecodes, syncenter and syncexit bytecodes are guaranteed to be hierarchically nested because they result from the invocation inlining tree. It is not possible, in Java, to execute the sequence: call foo(), call inner_foo(), return from foo(), return from inner_foo(). The syncenter/syncexit bytecodes for the foo() method must therefore execute around the syncenter/syncexit bytecodes for the inner_foo() method. That is, the artificial bytecodes may not be interleaved. The second main benefit is that, since these artificial bytecodes cannot occur in known Java class files, the JIT compiler recognizes that every syncexit bytecode must be balanced by an earlier syncenter bytecode. Only through an exception that is not handled in the method could a syncexit execute without the corresponding syncenter having executed earlier. Additionally, only through an exception that is not handled in the method could a syncenter acquire a lock that will not be later released by a corresponding syncexit. However, these exceptional cases may be correctly handled by a Java Virtual Machine (the environment in which such code is executed) that is aware of the syncenter and syncexit bytecodes.

These two benefits permit a much more efficient mechanism so that, when an exception occurs, all objects that are locked within the try block are properly unlocked. The first benefit (the guarantee of hierarchical nesting) allows the JIT compiler to dedicate contiguous and small sets of storage locations in the stack frame and to store,

in each set, references to objects that have been locked within the method via the syncenter bytecode. In the Java Virtual Machine, as each syncenter bytecode executes, a reference to the object that is being locked is placed into one specific storage location in the stack frame. The storage location is determined by the depth of the syncenter bytecode in a tree of inlined invocations for this method.

FIG. 4 illustrates a synchronization depth table 400 for the inlined go_to_it() method 200 illustrated in FIG. 2. The synchronization depth table 400 is used to associate a synchronization depth with ranges of program counter addresses. FIG. 5 illustrates a stack frame 500 associated with the inlined go_to_it() method 200. A corresponding inlining tree 600, also associated with the inlined go_to_it() method 200, is illustrated in FIG. 6. FIG. 7 illustrates simplified code 700 for the inlined go_to_it() method 200, showing only the synchronization operations.

The herein proposed JIT compiler initially performs a single pass over the code of a method to compute a synchronization depth at the beginning of each basic block of code. These basic block synchronization depths are then used by the JIT compiler during code generation to determine the ranges of program counter addresses in the method that have the same synchronization depth. These ranges are stored in the synchronization depth table 400 illustrated in FIG. 4, which may be efficiently searched to determine the synchronization depth at any particular program counter address when an exception occurs.

In operation, described in conjunction with a compiler flow diagram illustrated in FIG. 8, the JIT compiler considers each instruction (step 802) of each block of a method and generates executable code (step 804) corresponding to each instruction. While doing so, the JIT compiler maintains an indication of the current synchronization depth. Where the synchronization depth determined (step 806) for the current instruction is determined (step 808) to differ from the synchronization depth determined for the directly preceding instruction, the synchronization depth determined for the

directly preceding instruction is stored in the synchronization depth table 400 associated with a range of program counter addresses that share the same synchronization depth (step 810). The range includes the program counter address of the directly preceding instruction.

The basic blocks of the method code are not necessarily considered in order. That is, a current basic block of code under consideration may not be the direct successor to the previous basic block of code considered by the JIT compiler. As such, information collected in the hereinbefore discussed pass over the code to compute a synchronization depth at the beginning of each basic block of code is used to properly set the maintained indication of synchronization depth as the JIT compiler begins to consider a particular basic block of code.

In the illustrated example, such determining for the inlined go_to_it() method 200 results in the synchronization depth table 400 illustrated in FIG. 4. Notably, the synchronization depth table 400 associates ranges of program counter addresses in the method with a particular synchronization depth. Clearly, it may be desirable to conserve memory by not storing the ranges for which no synchronization depth exists. As the table stores the ranges of program counter addresses sequentially, it will be understood by a program accessing the synchronization depth table 400 that, if a given program counter address is not found, no synchronization depth exists for the given program counter address.

It may be seen that, in the simplified code 700 (FIG. 7), the foo() method is the topmost inlined synchronized method in the go_to_it() method 200. As such, when executed, the syncenter bytecode for the foo() method (line 702) causes a reference to an object required to be locked to be stored at the "depth 0" location in the stack frame 500 (see FIG. 5) associated with the inlined go_to_it() method 200. Similarly, execution of the syncenter bytecode for the inner_foo() method (line 703) causes a reference to an object required to be locked to be stored at the "depth 1" location. Execution of the

syncexit bytecode for the inner_foo() method (line 704) causes the reference to the object at the "depth 1" location to be read and the referenced object is then unlocked. When the syncexit bytecode for the foo() method (line 705) executes, the reference to the locked object at the "depth 0" location is read and the referenced object is then unlocked. Subsequently, execution of the syncenter bytecode for the bar() method (line 706) causes a reference to an object required to be locked to be stored at the "depth 0" location. When the syncexit bytecode for the bar() method (line 707) executes, the reference to the object at the "depth 0" location is read and the referenced object is then unlocked.

From the foregoing, it will be appreciated that the synchronization depth table 400 of FIG. 4 is created as the code is compiled and that the compiled code causes storing of object references to, and the reading of object references from, the stack frame 500 of FIG. 5 while the compiled code executes.

In view of FIG. 9, exception handling begins when an exception handler recognizes that a exception has been thrown (step 902), say during the execution of the inlined go_to_it() method 200. Responsive to such a recognition, the exception handler determines a program counter address at which the exception occurred (step 904). The synchronization depth table 400 (FIG. 4) is then consulted to determine, given the program counter address at which the exception occurred, the synchronization depth (step 906). For instance, should an exception be thrown during the execution of line 214 of the inlined go_to_it() method 200, the range 214-215 of the synchronization depth table 400 is consulted to determine that the synchronization depth is depth 0. The handler may then use the synchronization depth to read the appropriate reference to an object to be unlocked (step 908). The reading is performed, in this example, from the stack frame 500 associated with the inlined go_to_it() method 200. The reference to the locked object may then be obtained from the location in the stack frame 500 that is associated with depth 0. Once the reference to the object is obtained, the object may then be unlocked (step 910).

Inherently, a given exception handler is aware of the number of locks to be removed when an exception causes the given exception handler to be called. That is, the given exception handler is prearranged to be called at a preset synchronization depth. By consulting the synchronization depth table 400, the given exception handler determines the current synchronization depth. The difference between the current synchronization depth and the preset synchronization depth provides an indication to the given exception handler of the number of objects that are to be unlocked before the given exception handler can execute normally.

Without such a technique, many more handlers and many more exceptions may be required to unlock the correct number of objects when an exception occurs. If an exception occurs at line 310 in the typical compiled code listing 300 (FIG. 3), then two additional handlers must execute to unlock the appropriate objects (i.e., three exceptions are thrown and handled). In a version of the corresponding inlined go_to_it() method 200 illustrated in FIG. 2 compiled according to herein presented methods, if the exception happens at the same place (line 208) then only one exception is thrown and handled.

Moreover, known JIT compilers cannot optimize the Java code as well as the herein proposed JIT compiler in the presence of the extra try regions required to unlock each object. Aspects of the present invention allow the herein proposed JIT compiler to generate better code while accelerating exception handling.

Advantageously, aspects of the present invention allow the number of objects that must be unlocked when an exception occurs to be efficiently determined by a single, fast lookup in a table of program counter address ranges for the current method.

Other modifications will be apparent to those skilled in the art and, therefore, the invention is defined in the claims.